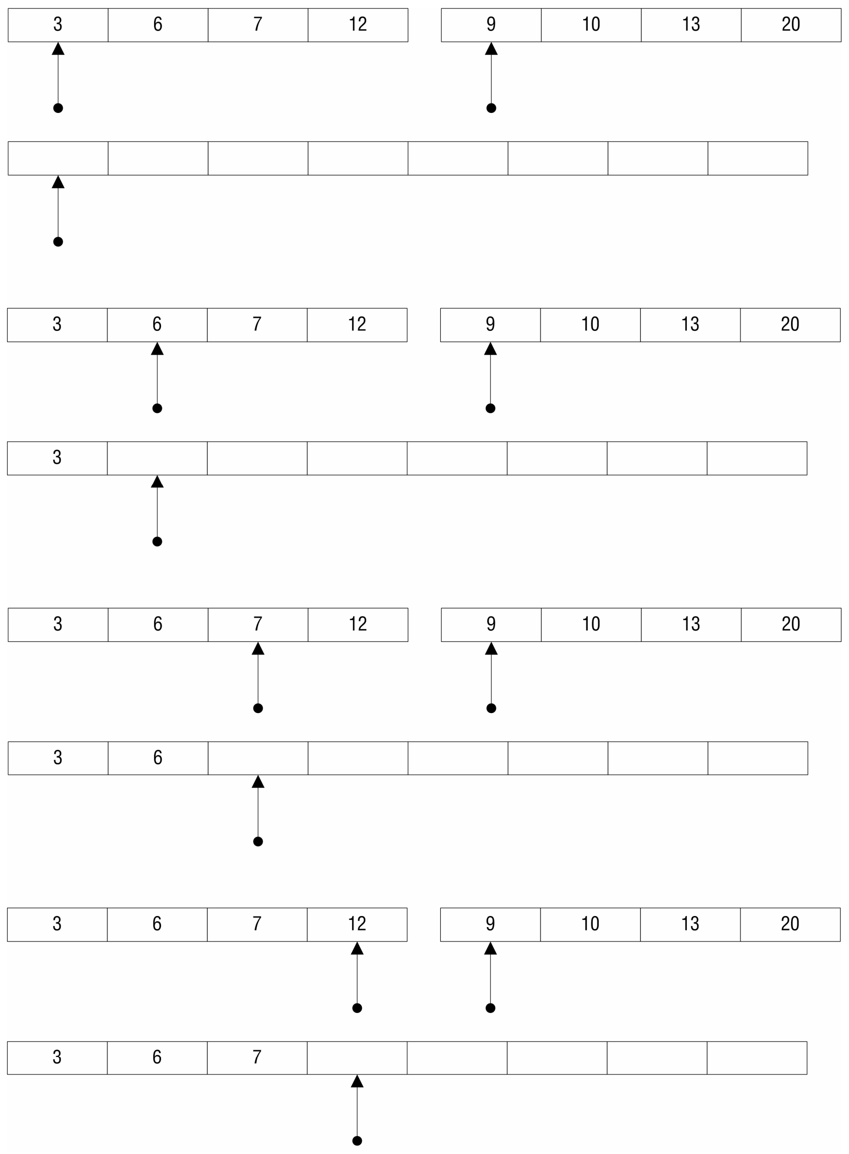
Chapter 9 Vector Sorting Revisited

We now return to sorting. In this chapter we will examine two sorts with n lg(n) average-case running times, the *merge sort* and the *quick sort*, both of which are typically implemented recursively. We will also discuss another quadratic sort, the *insertion sort*, and a subquadratic generalization of it called *Shell's sort*.

9.1 The Merge Sort Algorithm and Its Time Analysis

The merge sort is a divide-and-conquer algorithm whose fundamental operation is the *merging* of two sorted subarrays. The algorithm first uses recursive calls to subdivide the target array until pairs of one-element subarrays remain. Each pair of one-element subarrays is then merged, yielding half as many sorted two-element subarrays. Pairs of sorted two-element subarrays are then merged to create half as many sorted four-element subarrays, and so on, until two sorted subarrays of (approximate) size $\frac{n}{2}$ are left. These two subarrays are then merged to give the final sorted array.

The merging of two sorted arrays is straightforward. The figure below illustrates the merging of two four-element arrays. Note the use of a pointer to mark the current location in each array; at each stage the smaller of the two items of interest is copied into the next available location of a third array.


3 6 7 12 9 10 13 20
3 6 7 12 9 10 13 20
3
3 6 7 12 9 10 13 20
3 6
3 6 7 12 9 10 13 20
3 6 7

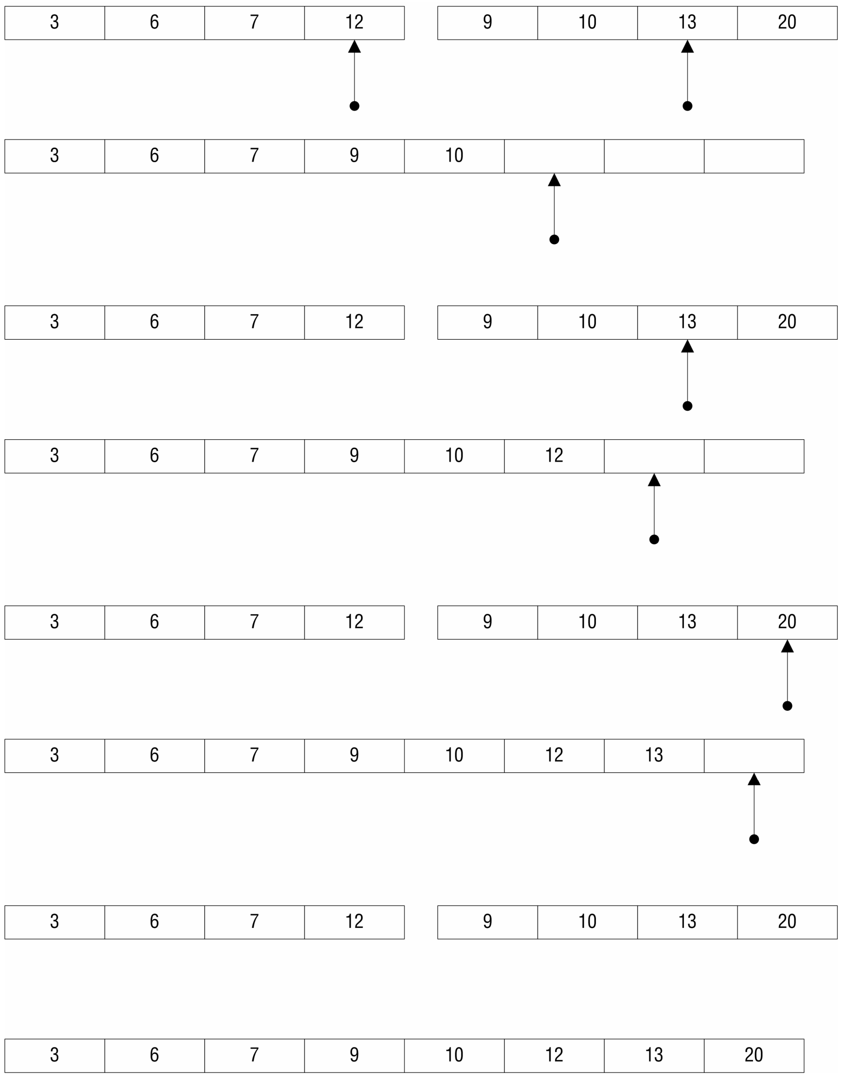


Fig. 9.1. The merging of two sorted four-element arrays

9.1.1 Merge Sort Implementation

The code for method `Sort.mergeSort` appears below.

```
public static void mergeSort(Vector vec, Comparable[] temp,
                             int left, int right)
{
    if (left == right)
        return;
    int mid = (left + right) / 2,
        i, j, k;
    mergeSort(vec, temp, left, mid);       // Divide and conquer.
    mergeSort(vec, temp, mid + 1, right);

    // The following loop copies the elements of the two adjacent
    // subvectors to 'temp'.

    for (j = left; j <= right; j++)
        temp[j] = (Comparable)vec.elementAt(j);
    i = left;
    k = mid + 1;

    // The following loop merges the two subarrays into the
    // target vector.

    for (j = left; j <= right; j++)
        if (i == mid + 1)
            vec.replace(j, temp[k++]);
        else if (k > right)
            vec.replace(j, temp[i++]);
        else if ((temp[i]).compareTo(temp[k]) < 0)
            vec.replace(j, temp[i++]);
        else
            vec.replace(j, temp[k++]);
}
```

9.1.2 Analysis of the Merge Sort

The outer call of `mergeSort` first divides the target array of size n in half, and then it makes a recursive call on each half. Each of those calls in turn divides its subarray of size $\frac{n}{2}$ in half before making two more recursive calls on subarrays of size $\frac{n}{4}$. This process continues until the target array cannot be further subdivided, i.e., until the target array has been divided into subarrays of size one. Thus approximately $\lg(n)$ subdivisions occur.

For each stage of subdivision, merges are performed, first on pairs of one-element subarrays, then on pairs of two-element subarrays, and so on, until the outer call merges a pair of $\frac{n}{2}$-element subarrays. The merges performed at each stage are collectively linear in their running time, for every element of the target array gets copied first to the temporary array and then back to the target array during some merge. Thus the merge sort carries out approximately lg(*n*) merges, each of which performs 2*n* operations. The merge sort is therefore in O(*n* lg(*n*)), with respect to running time, in the best, average, and worst cases.

9.2 The Quick Sort Algorithm and Its Time Analysis

The quick sort is the fastest known comparison-based sorting algorithm, in the average case. The quick sort is, like the merge sort, a divide-and-conquer algorithm. In the case of the quick sort, however, the fundamental operation is not merging but *partitioning*.

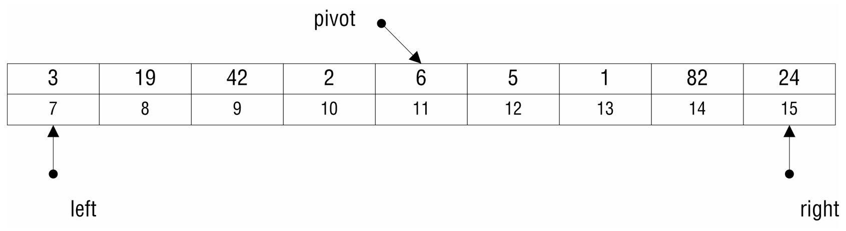
set partition: a collection of disjoint subsets of a set *S*, the union of which is equal to *S*

The quick sort partitions a subarray into two disjoint, adjacent portions. Each element in one portion is less than or equal to a value called the *pivot*, and each element in the other portion is greater than or equal to the pivot. The pivot value is usually chosen pseudorandomly or using a simple algorithm called *median-of-3*. Our implementation will use the latter method.

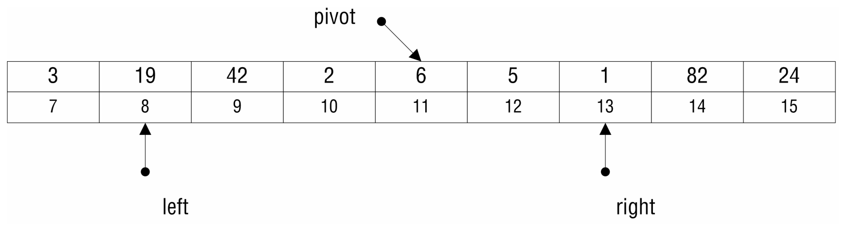
```
/*
'left' and 'right' mark the bounds of the subarray of interest.
Method medianOf3() sorts the left, middle, and right elements
of the subarray. The resulting middle element is the pivot, hence
the name median-of-3.
*/

protected static void medianOf3(Vector vec, int left, int right)
{
    int middle = (left + right) / 2;
    if (((Comparable)vec.elementAt(left)).compareTo(vec.elementAt(middle)) > 0)
        swap(vec, left, middle);
    if (((Comparable)vec.elementAt(middle)).compareTo(vec.elementAt(right)) > 0)
        swap(vec, middle, right);
    if (((Comparable)vec.elementAt(left)).compareTo(vec.elementAt(middle)) > 0)
        swap(vec, left, middle);
}
```

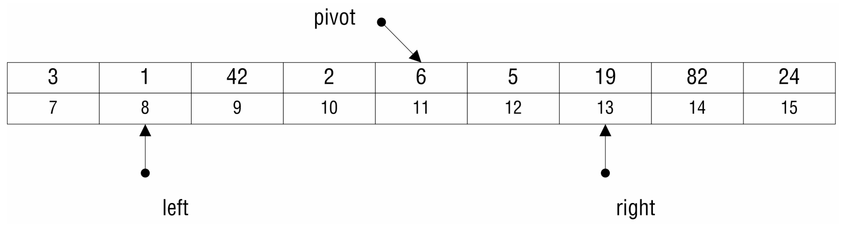
After the pivot has been chosen, partitioning begins. Let us step through the partitioning of the subarray shown below, assuming that `medianOf3` has already been called.



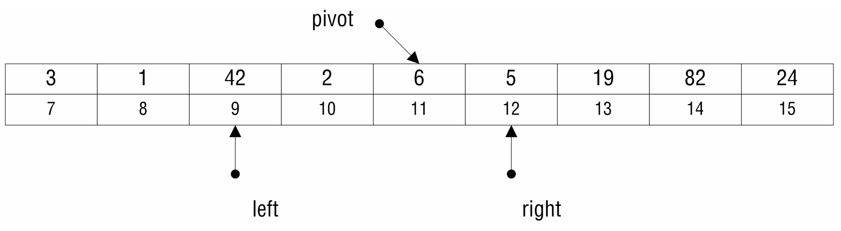
Left and right pointers are first placed at the bounds of the subarray. The left pointer is then moved to the right until it encounters a value that is greater than the pivot. In this case, that value is 19. After the left pointer stops, the right pointer is moved to the left until it encounters a value that is less than the pivot. In this case, that value is 1. The current state of affairs is shown below.



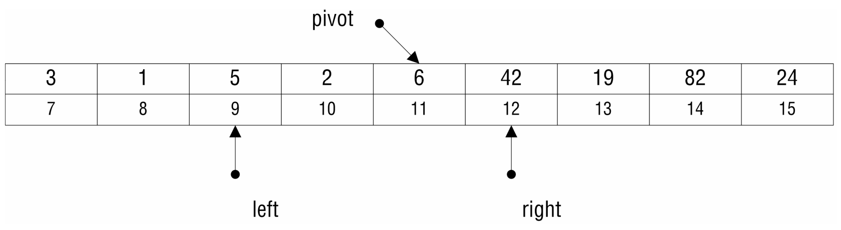
If the two pointers have not crossed, the two marked values are swapped. Thus the 19 and the 1 are exchanged, leaving the subarray as it appears below.



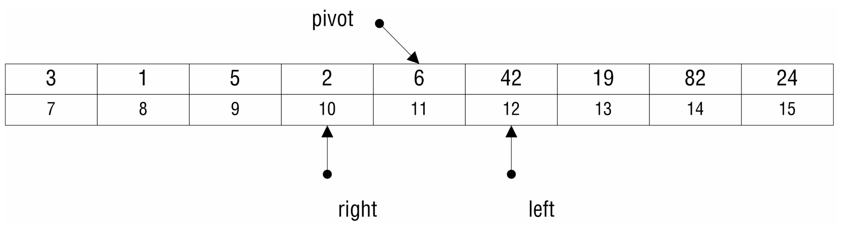
After the swap, the two pointers are again moved. This time the left pointer stops at 42, and the right pointer stops at 5.



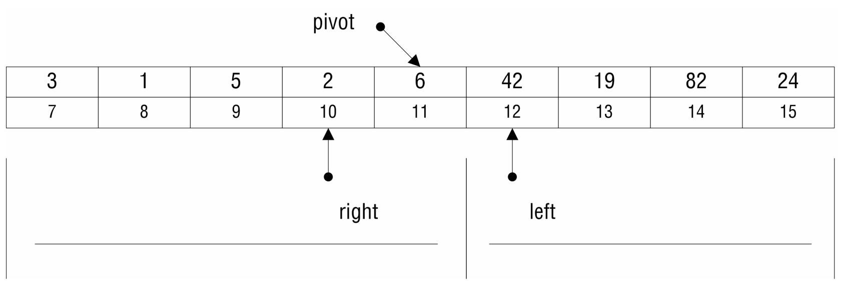
Since the pointers have not yet crossed, the 5 and the 42 are swapped, yielding the subarray shown below.



The left and right pointers are again moved, this time stopping at 42 and 2, respectively.



But now both pointers have crossed over the pivot, and so no swap is done. At this point we have a partitioned subarray, as shown below; all values at indices 12 through 15 are greater than or equal to the pivot, and all values at indices 7 through 10 are less than or equal to the pivot.



After partitioning the subarray, the quick sort makes a recursive call on each portion of the subarray. Each recursive call chooses a pivot from its portion and partitions that portion. This process continues until the size of a given subarray is smaller than some threshold value, in which case a helper sort, usually the insertion sort, is called upon to finish the job.

9.2.1 Quick Sort Implementation

The code for helper method `partition` is shown below, along with method `quickSort`.

```
protected static int partition(Vector vec, int left, int right)
{
    Object pivot = vec.elementAt((left + right) / 2);
    while (true)
    {
        while (((Comparable)vec.elementAt(++left)).compareTo(pivot) < 0);
        while (((Comparable)vec.elementAt(--right)).compareTo(pivot) > 0);
        if (left > right)
            break;
        else
            swap(vec, left, right);
    }
    return left;
}

public static void quickSort(Vector vec, int left, int right)
{
    if (right - left + 1 <= 10)          // If the subarray has
        insertionSort(vec, left, right); // fewer than 11 items,
    else                                 // then let insertionSort
    {                                    // finish the sorting.
        medianOf3(vec, left, right);
        int leftPtr = partition(vec, left, right);
        quickSort(vec, left, leftPtr - 1);
        quickSort(vec, leftPtr, right);
    }
}
```

9.2.2 Analysis of the Quick Sort

The analysis of the quick sort is similar to that of the merge sort but is complicated somewhat by the fact that the quick sort does not always partition a subarray into portions of equal, or nearly equal, size.

In analyzing the merge sort, we saw that the algorithm always divides a subarray in half, which can be done approximately lg(n) times for a target array of size n. The same is true of the quick sort, provided that suitable pivots are chosen. Thus, since the partitionings at each stage are collectively linear in their running time, as the reader should verify, the quick sort is in O(n lg(n)) in the best case.

The quick sort's worst-case performance occurs when each choice of pivot causes its subarray, say of size k, to be partitioned into a portion of size one and a portion of size $k - 1$. When this happens for each call to `quickSort`, the target array of size n gets subdivided n times, putting the quick sort in O(n^2) in the worst case.

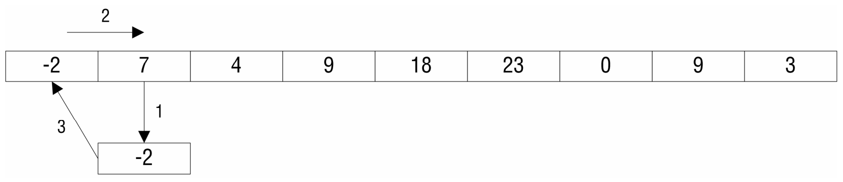
The average-case analysis of the quick sort requires more sophisticated techniques. Thus we will omit the average-case analysis and merely give the result: The quick sort is in O(n lg(n)) in the average case.

9.3 The Insertion Sort Algorithm and Its Time Analysis

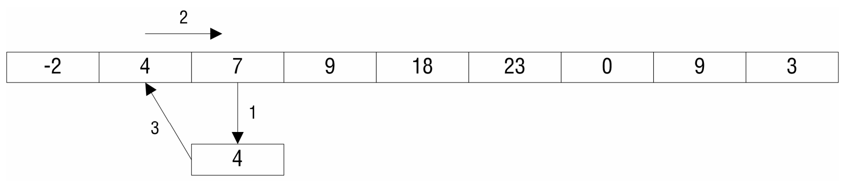
The insertion sort is in O(n^2) in the worst and average cases but is linear in the best case, which is why we use it as a helper for the quick sort. Let us apply the insertion sort to the example array shown below.

7	-2	4	9	18	23	0	9	3

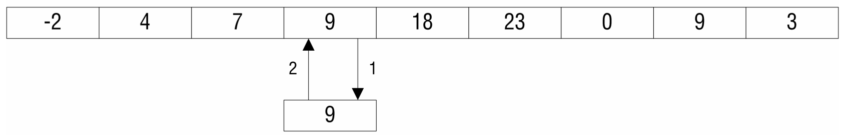
The insertion sort first "picks up" a target element, in this case the -2. The algorithm then shifts prior array elements to the right until it encounters an element that is less than the target element, or until the first array element has been shifted. Thus the 7 gets shifted to the right. Since 7 was the first element, no more shifting is done. The target element is then placed in the vacant slot, as illustrated by the next figure.



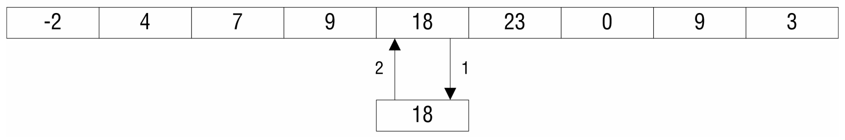
Now the target element is the 4. The 4 gets copied to a temporary holding place, then shifting begins. The 7 gets shifted again before the algorithm encounters the -2. Since -2 is less than the target, the -2 remains in the first array location and the 4 is placed just after it.

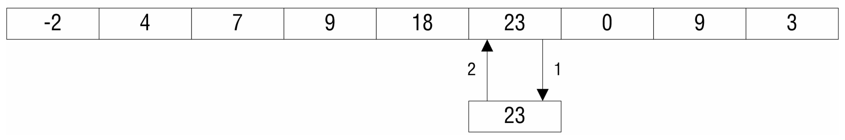


The next target element is the 9. The 9 stays put because it is already in the correct location relative to the first three elements.

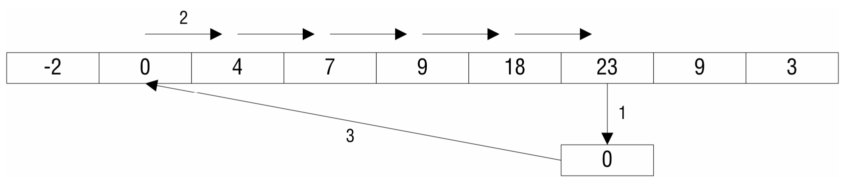


The next two target elements, 18 and 23, also need not be moved.

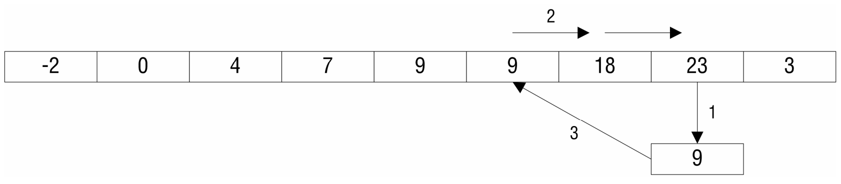




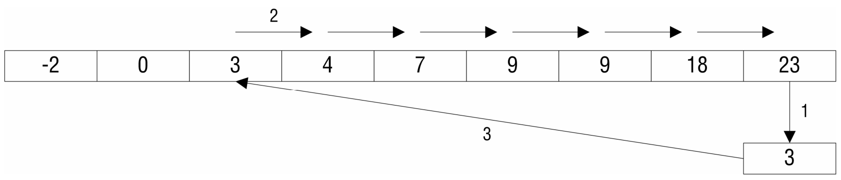
The next target element, 0, belongs just after the -2. Thus 23, 18, 9, 7, and 4 must be shifted to the right to make room for the 0, as shown below.



The second occurrence of 9 should of course be placed just after the first, and so 23 and 18 get shifted before the 9 is written to its correct location.



The final target element is the 3. Its final resting place should be just after the 0. Thus elements 23, 18, 9, 9, 7, and 4 are shifted, vacating the third array location. The 3 is then written to the array, completing the sort.



9.3.1 Insertion Sort Implementation

The insertion sort implementation is shown below.

```
public static void insertionSort(Vector vec, int left, int right)
{
    int inner,
        outer;
    Object target;
    for (outer = left + 1; outer <= right; outer++)
    {
        target = vec.elementAt(outer);// Move vec.elementAt(outer)
        inner = outer;                // to 'target'.

        /*
        While vec.elementAt(inner - 1) is greater than 'target',
        move vec.elementAt(inner) up one location. This is making
        room for 'target', so that it may be inserted at its
        correct location relative to the elements already
        processed.
        */

        while (inner > left &&
            ((Comparable)vec.elementAt(inner - 1)).compareTo(target) > 0)
        {
            vec.replace(inner, vec.elementAt(inner - 1));
            inner--;
        }
        vec.replace(inner, target); // Put 'target' at the correct
    }                               // location.
}
```

9.3.2 Analysis of the Insertion Sort

For an input size of n, the insertion sort inserts each of $n - 1$ elements at its correct location with respect to all elements that have already been processed. In the worst case the first insertion will require one comparison, the second insertion will require two comparisons, the third insertion will require three comparisons, ... , and insertion $n - 1$ will require $n - 1$ comparisons. Thus the total number of comparisons, in the worst case, will be

$$1+2+3+...+(n-3)+(n-2)+(n-1)=\frac{n(n-1)}{2}=\tfrac{1}{2}n^2-\tfrac{1}{2}n.$$

The insertion sort is therefore in $O(n^2)$ in the worst case.

We may safely assume that the algorithm will compare about half as often in the average case, making the number of comparisons approximately $\tfrac{1}{4}n^2-\tfrac{1}{4}n.$ Thus the insertion sort is in $O(n^2)$ in the average case.

In the best case, i.e., when the array is already sorted, only one comparison is required for each execution of the outer loop. Thus $n - 1$ comparisons are performed, which makes the insertion sort linear in the best case. If the array is only slightly disordered initially, we can expect the insertion sort's performance to be very nearly best case, which makes it a suitable helper for the quick sort. The idea is to let the quick sort do most of the work before calling upon the insertion sort to finish the job on small, nearly sorted subarrays. This technique boosts the quick sort's performance by eliminating a potentially large number of recursive calls.

9.4 Shell's Sort

The Shell sort algorithm, named after its discoverer, D.L. Shell, is a generalization of the insertion sort that can achieve subquadratic performance. The key concept is that of an *increment*, which we shall denote as h. The Shell sort chooses an initial value for h from an infinite sequence of increments, based on the size of the array to be sorted. The algorithm then *h-sorts* the array before resetting h to the preceding value from the sequence; to h-sort an array is to apply the insertion sort to each subset of array elements whose members are separated by a distance of h. Because the first value of an increment sequence is always 1, the final pass of the Shell sort is equivalent to the insertion sort.

Shell proposed the increment sequence 1, 2, 4, 8, ..., 2^{k-1}, 2^k, ... This sequence, however, yields poor performance relative to the sequence 1, 4, 13, 40, ... Using this latter sequence, the Shell sort is in O($n^{1.5}$) in the average case. Other proposed sequences represent modest improvements to this performance.

A Shell sort implementation follows.

```
public static void ShellSort(Vector vec)
{
    int inner, outer;
    Object target;
    int h = 1;
    while (h <= vec.size() / 3) // Calculate the initial
        h = 3 * h + 1;          // increment.
    while (h > 0)   // While the increment is greater than zero...
    {
        for (outer = h; outer < vec.size(); outer++)
        {
            target = vec.elementAt(outer);
            inner = outer;
            while (inner > h - 1 &&
            ((Comparable)vec.elementAt(inner - h)).compareTo(target) > 0)
            {
               vec.replace(inner, vec.elementAt(inner - h));
               inner -= h;
            }
            vec.replace(inner, target);
        }
        h = (h - 1) / 3;   // Calculate the new increment.
    }
}
```

Exercises

1. Implement the randomized quick sort, which chooses pivots pseudorandomly. Code an application that performs an empirical analysis of your two quick sort implementations.
2. Our quick sort implementation uses 10 as its threshold value for calling insertion sort on a subarray. Code an application that performs an empirical analysis of quick sort implementations with varying thresholds.
3. Apply the Shell sort, as it is implemented above, to an example array with at least 30 elements.